

Anomaly Sequences Detection from Logs Based on Compression

Wang Nan, Han Jizhong, and Fang Jinyun

Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China

Mining information from logs is an old and still active research topic. In recent years, with the rapid emerging of cloud computing, log mining becomes increasingly important to industry. This paper focus on one major mission of log mining: anomaly detection, and proposes a novel method for mining abnormal sequences from large logs. Different from previous anomaly detection systems which based on statistics, probabilities and Markov assumption, our approach measures the strangeness of a sequence using compression. It first trains a grammar about normal behaviors using grammar-based compression, then measures the information quantities and densities of questionable sequences according to incrementation of grammar length. We have applied our approach on mining some real bugs from fine grained execution logs. We have also tested its ability on intrusion detection using some publicity available system call traces. The experiments show that our method successfully selects the strange sequences which related to bugs or attacking.

PACS numbers:

I. INTRODUCTION

From trend analysing to system tuning, log mining technique is widely used in commercial and research area. In recent years, diagnosing systems according to logs becomes a hot research topic because of the rapid emerging of cloud computing systems¹⁻⁵. Problems in such systems are always non-deterministic because they are caused by uncontrollable conditions. Therefore, developers can watch neither the execution paths nor the communications between different components as they used to be. The only cute can be used are the text logs generated by buggy systems. However, the huge size of such logs makes developers hard to deal with them.

Aiming at non-deterministic problems, many approaches are proposed. Record-replay⁶⁻⁹ is a hopeful one. Record-replay systems record low level execution detail during running. When debugging, they can replay the buggy execution according to those data, let developers to check control flow and data flow of the target programs. Nevertheless, although recent record-replay tools can achieve low performance impact to be suitable for deploying into production environments^{7,8}, replaying 7x24 long lasting logs and manually identifying the key parts of the execution flow is still an obstacle.

The heart of the above problems is finding unusual patterns from large data set. This is the goal of intrusion detection. There are 2 general approaches of intrusion detection: misuse intrusion detection (MID) and anomaly intrusion detection (AID). MID models unusual behaviors as specific patterns and identifies them from logs. However, MID systems are vulnerable against unknown abnormal behaviors. This paper focus on AID, which models normal behaviors and reports unacceptable deviations. Anomaly detection has already been studied for decades. The related techniques are used for detection of network intrusion and attacking¹⁰⁻¹³. Those approaches apply probability models and machine learning algorithms¹¹⁻¹³, most of them rely on Markov assumption. They have achieved positive results on some spe-

cific type of logs such as system call traces^{11,13,14}. However, although Markov assumption makes them sensitive to unusual state transitions at low level, they are short to identify high level misbehavior.

This paper proposes a novel anomaly detection method. Different from the above approaches, our method doesn't rely on statistics, probabilities or Markov assumption, and needn't complex algorithms used in machine learning. The principle of our approach is straightforward: using compression to measure the information quantities of sequences. Our method can be used to find some high level abnormal behavior. To the best of our knowledge, our work is the first attempt to utilize the relationship between information quantities and compression in mining unusual sequences from logs.

We first introduce the principle of our approach using a simple example in section II, then present the detail algorithms in section III. Section IV lists a set of experiments to show the ability of our method on bug finding and intrusion detection. Section V concludes the paper.

II. OVERVIEW

Our approach is inspired from following obvious fact.

When the normal behavior of a information source is known to the viewer, she can describe another normal sequence use only a few words, but needs more words to describe an abnormal sequence. For example, the viewer is told that "1234 1235 1234 1235" are 4 normal sequences. For a questionable sequence "1235", she can describe it as "another type II sequence". The information contained in her description is only the "type II". However, for sequence "1237", the most elegant description should be "replace the forth character of normal pattern by 7". The information contained in this description is "forth" and "7". For sequence "32145", she has to say "a new sequence, the first character is 3, the second character is 2 ...". The information contained in this description is much more than the previous two. Therefore, the viewer

TABLE I: The size of gzipped data

n	evaluated sequence	gzipped sequence	Q_n	I_n
0		1234123512341235	30	0
1	1234	12341235123412351234	30	0
2	1237	12341235123412351237	31	1
3	32145	12341235123412353214535	5	

can infer that the last sequence is “*the most strange one*”. An anomaly detection system should report the last sequence among the three.

In computer science there is a method to “describe a sequence”: compression. A compression algorithm can reduce the size of a sequence. For a long sequence, the compressed data can be thought as “the description of the original sequence”. It is well known that no compression algorithm is able to ultimately reduce the size of a sequence to zero. It is also well known that a compressed file is hard to be compressed again, entropy rate ($H(X)$) of the source data restricts the performance of a compression algorithm.

Our approach utilizes the relationship between the information quantity and the compressed data size. To select the “*most strange*” sequence, we can use the following 3 steps:

- Training: compress a set of normal sequences, the compressed data size is Q_0 .
- Evaluating: for each candidate sequence, add it into the normal set used in training step, compress the new set. The compressed size is Q_n . Let $I_n = Q_n - Q_0$.
- Selecting: the n th sequence which generates the largest Q_n is selected.

We use the previous example to demonstrate the above 3 steps. The compression algorithm is **gzip**.

The first row of table I shows the result of training. Sequence *1234123512341235* is combined from 4 normal sequences, gzip compresses it into 30 bytes. Following rows show the evaluating step. 3 questionable sequences are appended then gzipped. The incrementation of the three are 0, 1 and 5. The third step selects *32145* as “the most strange” one as it generates the largest I_n .

III. DETAIL

In this section we introduce our approach in detail.

A. Grammar-based compression algorithm

Although we use gzip algorithm to explain the principle of our anomaly detection method in section II, gzip and

other well known generic compression algorithms are not suitable for digging anomaly sequences from execution logs because of the following reasons:

- Nearly all well known compression algorithms (such as gzip, bzip2 and rar) are based on LZ77¹⁵, which uses sliding window to store recent data for matching incoming stream and ignore previous data. Sliding window is important for a generic compression algorithm for compressing speed. However, it eliminates the historic knowledge about the data source, makes those data effectless when compressing new data.
- Generic compression algorithms compress data as byte stream. Their alphabets are 256 possible bytes from 0x0 to 0xff. However, the unit of execution logs is log entry. A compression algorithm with alphabet made by possible entries can discover meaningful patterns.
- Generic compression algorithms are unable to identify difference sequences when training and evaluating. Sequences in execution logs have to be stick together one by one. Some patterns will be created unintentionally across different sequences and affect the evaluating processing. In previous example, generic compression algorithms take pattern “*34123*” into account although it is not a part of any sequences.

Our approach chooses a grammar-based codes as the underlying compression algorithm. Grammar-based compression algorithms are developed recent decades as a new way to losslessly compress data^{16–19}. Kieffer et al.¹⁶ firstly published some important theorems on it. This paper uses same symbols and terminologies to describe the algorithm. Yang et al.¹⁷ presented a greedy grammar transform. Our algorithm is based on it. Such grammar transform is similar to SEQUITUR^{18,19}, but generates more compact grammars.

The idea of grammar-based compression is simple: for stream x , one can represents it as a context-free grammar G_x which generates language $\{x\}$ and takes much less space to store. Grammar-based compression is suitable for compressing execution logs because such logs are generated by program hierarchically and are highly structured.

1. Grammar transform overview

Grammar transform converts a sequence x into an admissible grammar G_x that represents x . An admissible grammar G_x is such a grammar which guarantees language $L(G) = \{x\}$. (The language of G_x contains only x .) We define $G = (V, T, P, S)$ in which

- V is a finite nonempty set of *non-terminals*.
- T is a finite nonempty set of *terminals*.

- P is a finite set of *production rules*. A production rule is an expression of the form $A \rightarrow \alpha$, where $A \in V$, $\alpha \in (T \cup V)^+$. (α is nonempty).
- $S \in V$ is *start symbol*.

Define f_G to be endomorphism on $(V(G) \cup T(G))^*$ such that:

- $f_G(a) = a$, $a \in T(G)$
- $f_G(A) = \alpha$, $A \in V(G)$ and $A \rightarrow \alpha \in P(G)$
- $f_G(\epsilon) = \epsilon$
- $f_G(u_1 u_2) = f_G(u_1) f_G(u_2)$

Define a family of endomorphism $\{f^k : k = 0, 1, 2, \dots\}$:

- $f_G^0(x) = x$ for any x
- $f_G^1(x) = f_G(x)$
- $f_G^k(x) = f_G(f_G^{k-1}(x))$

Kieffer et al. showed¹⁶ that, for an admissible grammar G_x , $f_{G_x}^{|V(G_x)|}(u) \in (T(G_x))^+$ for each $u \in (V(G_x) \cup T(G_x))^+$, and $f_{G_x}^{|V(G_x)|}(G_x(S)) = x$. Informally speaking, for an admissible grammar G_x , by iteratively replacing non-terminals with the right side of corresponding production rules, every $u \in (V(G) \cup T(G))^+$ will finally be translated into a string which contains only terminals. Define a mapping f_G^∞ such that $f_G^\infty(u) = f_G^{|V(G)|}(u)$ for each $u \in (V(G) \cup T(G))^+$. Informally speaking, $f_G^\infty(u)$ is the original sequence represented by u .

2. The greedy grammar transform algorithm

The algorithm we used is based on following reduction rules (in following description, α and β represent string in $(V(G) \cup T(G))^*$):

1. For an admissible grammar G , if there is a non-terminal A which appears at right side of production rules only once in $P(G)$, let $A \rightarrow \alpha$ be the production rule corresponding to A , let $B \rightarrow \beta_1 A \beta_2$ be the only rule which contain A in its right side, remove A from $V(G)$ and remove $A \rightarrow \alpha$ from $P(G)$, then replace the production rule of B by $B \rightarrow \beta_1 \alpha \beta_2$.
2. For an admissible grammar G , if there is a production rule $A \rightarrow \alpha_1 \beta \alpha_2 \beta \alpha_3$ where $|\beta| > 1$, add a new non-terminal B into $V(G)$ then create a new rule $B \rightarrow \beta$, replace the production of A by $A \rightarrow \alpha_1 B \alpha_2 \beta \alpha_3$.
3. For an admissible grammar G , if there are two production rules A_1 and A_2 that $A_1 \rightarrow \alpha_1 \beta \alpha_2$ and $A_2 \rightarrow \alpha_3 \beta \alpha_4$, in which $|\beta| > 1$ and either $|\alpha_1| > 0$ or $|\alpha_2| > 0$, either $|\alpha_3| > 0$ or $|\alpha_4| > 0$,

```
#Transform x into an admissible grammar
#returns the start rule by p0, other rules by G
def GrammarTransform(x):
    G = {}
    p0 = SeqTransform(x, G)
    return p0, G
```

```
#x is the sequence to be transform
#G is a set of production rules
#output: return the start symbol p_x so that f_G^\infty(p_x) = x,
# all other rules are added into G
def SeqTransform(x, G):
    p_x = S_x \rightarrow \epsilon
    while |x| > 0:
        #greedy read ahead and match
        for p in G:
            v = left side of p
            check whether f_G^\infty(v) is x's prefix
        if matched:
            v = the longest matched nonterminal
            append v after the right side of p_x
            pop |f_G^\infty(v)| entries from x
        else:
            pop one entry t from x
            append t as a terminal after the right side of p_x
            apply reduction rules 1-3 iteratively over G \cup \{p_x\},
            until non of them can be applied.
            newly created rules are added into G
    return p_x
```

FIG. 1: Greedy grammar transform algorithm

add a new non-terminal B into $V(G)$ then create a new rule $B \rightarrow \beta$, replace the production of A_1 by $A_1 \rightarrow \alpha_1 B \alpha_2$, replace the production of A_2 by $A_2 \rightarrow \alpha_3 B \alpha_4$.

Figure 1 illustrates the grammar transform algorithm¹⁷. It is very similar to SEQUITUR^{18,19} except the greedy read ahead step, which guarantees that in the generated grammar G , for different $v \in V(G)$, $f_G^\infty(v)$ are different.

Algorithm in figure 1 transforms a sequence into a context-free grammar. To avoid patterns across different sequences interfering the processing, we wrap the algorithm as figure 2. In the wrapped algorithm, we can guarantee that every execution sequences are represented by a non-terminal in the right side of p_0 . The reduction rules never consider patterns across sequences because p_0 is not in G . In figure 2 we also show that our algorithm eliminates redundant sequences by dropping those results which contain only one symbol.

In table II we explain the above algorithm using an example of computing a grammar for 4 sequences 1234 1235 1234 1237. The final grammar is listed at the last row in the table.

We measure the quantities of information of a sequence by computing the number of additional symbols which it introduces into the grammar. Figure 3 describes the evaluating process. After a grammar generated, **EvaluateSequence** is used to compute the information quantity (I) and information density D (average

```

# logs is a set of sequences
def LogTransform(logs):
    G = {}
    p0 = S0 → ε
    for seq in logs:
        pn = SeqTransform(seq, G)
        if pn contains only one symbol in its right side:
            drop pn
        continue
    insert pn into G
    append pn after the right side of p0
    return (p0, G)

```

FIG. 2: Transform a set of sequences

TABLE II: Example of 4 sequences: 1234 1235 1234 1237

processed p_n	G	p_0
string		
begin process sequence 1234		
$p_1 \rightarrow \epsilon$	{}	$p_0 \rightarrow \epsilon$
1234	$p_1 \rightarrow 1234$	
begin process a new sequence 1235		
$p_2 \rightarrow \epsilon$	$\{p_1\}, p_1 \rightarrow 1234$	$p_0 \rightarrow p_1$
12	$p_2 \rightarrow 12$	
apply rule 3 on pattern 12		
$p_2 \rightarrow p_a$	$\{p_1, p_a\}, p_1 \rightarrow p_a 34$	
	$p_a \rightarrow 12$	
123	$p_2 \rightarrow p_a 3$	
apply rule 3 on pattern $p_a 3$		
$p_2 \rightarrow p_b$	$\{p_1, p_a, p_b\}, p_1 \rightarrow p_b 4$	
	$p_a \rightarrow 12, p_b \rightarrow p_a 3$	
apply rule 1 on p_a		
	$\{p_1, p_b\}, p_1 \rightarrow p_b 4$	
	$p_b \rightarrow 123$	
1235	$p_2 \rightarrow p_b 5$	
begin process a new sequence 1234		
$p_3 \rightarrow \epsilon$	$\{p_1, p_b, p_2\}, p_1 \rightarrow p_b 4$	$p_0 \rightarrow p_1 p_2$
	$p_b \rightarrow 123, p_2 \rightarrow p_b 5$	
look ahead greedy match:		
p_1 and p_b matched, $ f_G^\infty(p_1) $ is the longest		
1234	$p_3 \rightarrow p_1$	
begin process a new sequence 1237		
p_3 contains only 1 symbol, eliminate p_3		
$p_4 \rightarrow \epsilon$		$p_0 \rightarrow p_1 p_2$
look ahead greedy match:		
p_b matched		
123	$p_4 \rightarrow p_b$	
1237	$p_4 \rightarrow p_b 7$	
finish processing		
	$\{p_1, p_b, p_2, p_4\}, p_1 \rightarrow p_b 4$	$p_0 \rightarrow p_1 p_2 p_4$
	$p_b \rightarrow 123, p_2 \rightarrow p_b 5$	
	$p_4 \rightarrow p_b 7$	

symbols produced by an entry) of a sequence x . To illustrates the evaluating process, we evaluate sequences 2238 and 1239 using G generated by table II.

From the above table, 2238 is more strange than 1239.

```

# Count the number of total symbols which is needed for
# describing all rules in rules
def EvaluateRules(rules, G):
    v = 0
    processedrules = {}
    for r in rules:
        if r is in processedrules:
            continue
        processedrules.insert(r)
        # r is a production rule with the form A → α
        v += |α|
    for symbol in α:
        if symbol is nonterminal:
            rn = G[symbol]
            if rn not in processedrules:
                rules.append(rn)
    return v

# x is the sequence which is to be evaluated
# p0 and G are parameters of an already computed grammar
def EvaluateSequence(x, p0, G):
    G' = G #deep copy
    pn = SeqTransform(x, G')
    info_old = EvaluateRules({p0}, G)
    info_new = EvaluateRules({p0, pn}, G')
    I = info_new - info_old
    D = I / |x|
    return I, D

```

FIG. 3: Evaluation of a new sequence

TABLE III: Evaluation of 2 sequences

G	2238	1239	
	$p_0 \rightarrow p_1 p_2 p_4$	$p_{2238} \rightarrow 2 p_c 8$	$p_{1239} \rightarrow p_b 9$
	$p_b \rightarrow 123$	$p_0 \rightarrow p_1 p_2 p_4$	$p_0 \rightarrow p_1 p_2 p_4$
	$p_1 \rightarrow p_b 4$	$p_b \rightarrow 1 p_c$	$p_b \rightarrow 123$
	$p_2 \rightarrow p_b 5$	$p_c \rightarrow 23$	$p_1 \rightarrow p_b 4$
	$p_4 \rightarrow p_b 7$	$p_1 \rightarrow p_b 4$	$p_2 \rightarrow p_b 5$
		$p_2 \rightarrow p_b 5$	$p_4 \rightarrow p_b 7$
		$p_4 \rightarrow p_b 7$	
12 symbols	16 symbols	14 symbols	
	$I = 4$	$I = 2$	
	$D = 1$	$D = 0.5$	

B. Anomaly detection based on compression

We introduced out anomaly detection algorithm in this subsection.

The goal of the algorithm is to find abnormal sequences in given logs. The input are two data sets. One set contains some normal sequences, the other set contains questionable sequences. From the later set our algorithm reports abnormal sequences.

The algorithm can be divided into following steps:

1. Training: transform the normal set S_n into an admissible grammar G with $S(G) = p_0$.
2. Evaluating: for each sequences t_n in questionable set S_q , compute (I_{t_n}, D_{t_n}) using $\text{EvaluateSequence}(t_n, p_0, G)$.
3. Reporting: report m_1 sequences which generates

```

...
main:/.../server.c:516
main:/.../server.c:536
main:/.../server.c:538
server_init:/.../server.c:170
server_init:/.../server.c:172
...

```

FIG. 4: Sample ReBranch trace

largest m_1 I_{t_n} , report m_2 sequences which generates largest m_2 D_{t_n} . m_1 and m_2 are configurable.

We report abnormal sequences according to both I and D because we believe they are both meaningful. A sequence x which generates large I_x indicates that there is no a similar sequence in S_n . However, if x is a very long sequence, the symbols used to describe x may be at very high level. Compare with a short sequence y with $I_y \cong I_x$, y is more valuable.

IV. EXPERIMENTAL ANALYSIS

A. Fine grained execution log

We tested the ability of our method on finding bugs in fine grained execution log. The data sets we used are generated using ReBranch⁹. ReBranch is a record-replay tool for debugging. It records the outcome of all branch instructions when running, and replay the execution according to these logs for debugging. We converted the traces into line number sequences. Figure 4 shows a piece of sample trace.

We tried our algorithm on finding two non-deterministic bugs in `lighttpd` (a light weight web server) and `memcached` (a key-value object caching system).

In `lighttpd` bug 2217²⁰, sometimes a few of CGI requests timeout. The bug is caused by a race condition: when a child process exits before the parent process is notified about the state of the corresponding pipe, the parent will wrongly remove the pipe from the event pool and never close the connection because it assumes the pipe still contain data.

In our experiment on `lighttpd` bug, we first collected a trace with 500 correct requests for training, then collected another trace with 1000 requests for testing. 2 of these 1000 requests timeout. Traces are pre-processed to be divided into sequences. During the pre-processing, signal handling are removed. A sequence begin at the entry of `connection_state_machine()` and end at the exit point of that function. After pre-processing, the normal trace contains 2501 sequences made by 3337395 entries, the questionable trace contains 4996 sequences made by 6661425 entries.

`memcached` bug 106²¹ is combined by 2 bugs. We first fixed a udp deadlock problem under the help of ReBranch. After that, when the cache server receives a

TABLE IV: Test result of ReBranch data sets

lighttpd		train result:	
		3337395 entries into 2793 symbols	
top most 5 I		top most 5 D	
$I_{654} = 41$		$D_{654} = 0.039653$	
$I_{3990} = 41$		$D_{3990} = 0.039653$	
$I_{1172} = 3$		$D_{655} = 0.019231$	
$I_1 = 1$		$D_{3991} = 0.019231$	
$I_2 = 1$		$D_{22} = 0.018868$	
memcached		train result:	
		862636 entries into 582 symbols	
top most 5 I		top most 5 D	
$I_{1237} = 27$		$D_{1237} = 0.031765$	
$I_{1608} = 27$		$D_{1608} = 0.031765$	
$I_{1609} = 27$		$D_{1609} = 0.031765$	
$I_1 = 1$		$D_1 = 0.009091$	
$I_2 = 1$		$D_6 = 0.009091$	

magic udp packet, some of following udp requests won't get reply. The problem is caused by incorrect state transfer. `memcached` uses a state machine when serving a request. The incorrect state transfer is `conn_read -> conn_closing`. The correct transfer sequence is more complex.

In `memcached` experiment, we first collected a trace with 1000 correct udp requests, then tried to identify 3 buggy requests out of 1003 new requests. As previous experiment, we split traces into sequences. A sequence begin at the entry point of `event_handler()` and end at the exit point of that function. After splitting, normal trace data set contains 1442 sequences made by 862636 entries; questionable trace contains 1609 sequences made by 883226 entries.

The results of the above 2 experiments are listed in table IV. In `lighttpd` experiment, our algorithm find 2 sequences (654 and 3990) with I and D quite larger than others. In `memcached` experiment, our algorithm find 3 strange sequences (1237, 1608 and 1609). We confirmed those sequences are correct ones (buggy ones) by manually replaying.

It is hard to detect `memcached` 106 bug using traditional Markov-based intrusion detection method because the misbehavior is at a very high level. Markov-based methods only consider the probabilities of one entry transfer to another entry. However, in this example, state transfer operation is implemented by many lines, each line transfer is valid. If developer know the distance between the key lines which represent a state transfer, higher order Markov model or n-gram model can be used. Nevertheless, for different program, developer have to manually adjust the length of sliding window. Furthermore, computing higher order model requires much more resources—always grows exponentially.

data set	traces	procs	entries	...
xlock-synth-unm	71	71	339177	229 2
xlock-intrusions	2	2	949	229 1
named-live	1	27	9230572	370 66
named-exploit	2	5	1800	370 5
				370 63
				...

FIG. 5: UNM data set sample and size

TABLE V: Test result of UNM data sets				
xlock	train result: 266563 entries into 6149 symbols			
top most 5 normal sequences		2 exploited sequences		
$I_5 = 776$	$D_{10} = 0.107226$	$I_{q_1} = 183$	$D_{q_1} = 0.372709$	
$I_8 = 89$	$D_1 = 0.049743$	$I_{q_2} = 176$	$D_{q_2} = 0.380952$	
$I_1 = 87$	$D_5 = 0.036998$			
$I_4 = 59$	$D_6 = 0.032590$			
$I_{10} = 46$	$D_4 = 0.026375$			
named	train result: 9215497 entries into 66148 symbols			
top most 5 normal sequences		5 exploited sequences		
$I_2 = 322$	$D_2 = 0.031078$	$I_{q_2} = 90$	$D_{q_3} = 0.311688$	
$I_3 = 4$	$D_3 = 0.003537$	$I_{q_5} = 70$	$D_{q_2} = 0.297030$	
$I_4 = 4$	$D_5 = 0.003537$	$I_{q_3} = 24$	$D_{q_5} = 0.291667$	
$I_5 = 4$	$D_4 = 0.002093$	$I_{q_1} = 1$	$D_{q_1} = 0.001681$	
$I_1 = 1$	$D_1 = 0.001681$	$I_{q_4} = 1$	$D_{q_4} = 0.001681$	

B. System call sequences

We used the data set published by the University of New Mexico¹⁴ to evaluate the ability of our algorithm on intrusions detection. The published data sets are system call traces generated using **strace**.

We applied our algorithm on **xlock** and **named** data sets. Figure 5 shows the size of those data and some sample entries in those traces. A trace entry contains two numbers, the left one is process id, the right one is the system call number. A trace in UNM data set contains many processes.

We use our algorithm to identify exploited processes. To achieve this, we splitted the original traces into system call sequences according to process id. The entries in each result sequences contain only the system call number. For **xlock**, we randomly selected 61 processes sequences for training then compared I and D of the other 12 sequences (10 normal, 2 exploited); for **named**, we chose 22 of normal sequences for training. The result is listed in table V.

In **xlock** result, information density (D) of the two exploited sequences are 2 times larger than the largest density in normal set. In **named** result, our algorithm

identified 3 strange sequences, information densities of them are at different order of magnitude. The last 2 sequences generate only 1 symbol ($I = 1$), indicates that same sequences have appeared in the training set at least once. After checking we found that those 2 processes are the parent processes used to setup daemons, none of them is target of attacks.

TABLE VI: Processing speed

data set	training		evaluating	
	time (s)	throughput (ent/s)	time (s)	throughput (ent/s)
lighttpd	90.4	36918.1	496.3	13422.2
memcached	10.1	85409.5	28.7	30817.4
xlock	237.1	1124.3	166.7	442.1
named	15742.6	585.4	189.9	89.2

C. Performance

Finally we list the throughput of our algorithm in table VI. It has been shown that SEQUITUR is a linear-time algorithm¹⁸. Our algorithm is similar to SEQUITUR except the read ahead matching. Such matching (match a long string against many shorter strings and find the longest match) can be optimized using a prefix tree.

V. CONCLUSION

In this paper we propose a novel anomaly detection algorithm by comparing the incrementation of compressed data length based on grammar-based compression. To the best of our knowledge, this is the first work which uses compression to measure the strangeness of sequences in anomaly detection. Different from Markov-based algorithm, our method utilizes the full knowledge about the structure of the data set. It can be used to find high level misbehavior as well as low level intrusions. We tested the algorithm on finding bugs in fine grained execution logs and intrusion detection in system call traces. In both data set, our method got positive result. The proposed method is also applicable to text log generated by today's cloud computing systems.

Acknowledgements This work is partially supported by the National Natural Science Foundation of China (Grant No. 61070028 and 61003063).

¹ D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. *ACM SIGPLAN Notices*, 45(3):143–154, 2010.

² D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In

Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, pages 3–14. ACM, 2011.

³ Cheng Zhang, Zhenyu Guo, Ming Wu, Longwei Lu, Yu Fan, Jiajun Zhao, and Zheng Zhang. AutoLog: Facing log redundancy and insufficiency. In *Proceedings of the 2nd*

- ACM SIGOPS Asia-Pacific Workshop on Systems. ACM, 2011.
- ⁴ J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Mochi: visual log-analysis based tools for debugging hadoop. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 18–18. USENIX Association, 2009.
 - ⁵ W. Xu, L. Huang, A. Fox, D. Patterson, and M.I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132. ACM, 2009.
 - ⁶ Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, 2008.
 - ⁷ Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*, pages 177–192, New York, NY, USA, 2009. ACM.
 - ⁸ Nan Wang, Jizhong Han, Haiping Fu, Xubin He, and Jinyun Fang. Reproducing non-deterministic bugs with lightweight recording in production environments. 2011.
 - ⁹ ReBranch: a debugging tool for replay bugs, <http://code.google.com/p/rebranch>, 2011.
 - ¹⁰ D.E. Denning. An intrusion-detection model. *Software Engineering, IEEE Transactions on*, (2):222–232, 1987.
 - ¹¹ Y. Qiao, X.W. Xin, Y. Bin, and S. Ge. Anomaly intrusion detection method based on hmm. *Electronics Letters*, 38(13):663–664, jun 2002.
 - ¹² S. Hyun Oh and W. Suk Lee. An anomaly intrusion detection method by clustering normal user behavior. *Computers & Security*, 22(7):596–612, 2003.
 - ¹³ Xinguang Tian, Xueqi Cheng, Miya Duan, Rui Liao, Hong Chen, and Xiaojuan Chen. Network intrusion detection based on system calls and data mining. *Frontiers of Computer Science in China*, 4:522–528, 2010. 10.1007/s11704-010-0570-9.
 - ¹⁴ C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 133–145. IEEE, 1999.
 - ¹⁵ J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.
 - ¹⁶ John C. Kieffer and E. Yang. Grammar-based codes: a new class of universal lossless source codes. *Information Theory, IEEE Transactions on*, 46(3):737–754, 2000.
 - ¹⁷ En-Hui Yang and John C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. i. without context models. *Information Theory, IEEE Transactions on*, 46(3):755–777, may 2000.
 - ¹⁸ Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.
 - ¹⁹ Craig G. Nevill-Manning and Ian H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2 and 3):103, 1997.
 - ²⁰ Race condition leaving response not forwarded to HTTP client, <http://redmine.lighttpd.net/issues/2217>, 2010.
 - ²¹ binary protocol parsing can cause memcached server lockup, <http://code.google.com/p/memcached/issues/detail?id=106>, 2010.